File I/O Lesson Outline

File I/O Using Redirection #1

So far in C, we've been using only standard input (keyboard) and standard output (monitor).

We know that we can input from a file by redirecting the file into standard input:

```
% big_statistics < actual_1.txt
```

Likewise, we can output to a file by redirecting standard output into a file:

```
% big_statistics > actual_1_output.txt
```

File I/O Using Redirection #2

In fact, we can combine redirected input with redirected output:

```
% big_statistics < actual_1.txt \
    > actual_1_output.txt
```

But what if we wanted to use multiple files at the same time?

For example, suppose we wanted to run a weather forecast, and we had a file containing our initial conditions (for example, the observed weather at midnight) and a different file containing data describing the terrain of the continental US.

We want to input from both of these files. But how?

Direct File I/O #1

Most programming languages support reading and writing files directly from within the program, without having to use redirecting.

In C, we can ***open*** a file using the ***`fopen`*** function:

```
...
char filename[filename_length+1];
FILE* fileptr = (FILE*)NULL;
...
strcpy(filename, "actual_1.txt");
fileptr = fopen(filename, "r");
...
```

What does this mean?

Direct File I/O #2

```
...
char filename[filename_length+1];
FILE* fileptr = (FILE*)NULL;
...
strcpy(filename, "actual_1.txt");
fileptr = fopen(filename, "r");
...
```

The ***`fopen`*** function opens a file in preparation for reading, writing or appending to a file.

The first argument is a string representing the filename.

The second argument is a string that encodes the ***I/O mode***.

File I/O Mode

```
...
char filename[filename_length+1];
FILE* fileptr = (FILE*)NULL;
...
strcpy(filename, "actual_1.txt");
fileptr = fopen(filename, "r");
...
```

The second argument is a string that encodes the I/O mode, which can be:

- `"r"` : Open the file for reading.
- `"w"` : Open the file for writing.
- `"a"` : Open the file for appending (writing at the end of an existing file).

FILE Pointer

```
...
char filename[filename_length+1];
FILE* fileptr = (FILE*)NULL;
...
strcpy(filename, "actual_1.txt");
fileptr = fopen(filename, "r");
...
```

The function **`fopen`** returns a ***file pointer***, which is a pointer to a special data type that's used to identify and describe a file.

FILE Pointer == NULL #1

```
...
char filename[filename_length+1];
FILE* fileptr = (FILE*)NULL;
...
strcpy(filename, "actual_1.txt");
fileptr = fopen(filename, "r");
...
```

The function **`fopen`** returns a ***file pointer***, which is a pointer to a special data type that's used to identify and describe a file.

If for some reason the file can't be opened, then the return value of `fopen` is `NULL`.

FILE Pointer == NULL #2

```
fileptr = fopen(filename, "r");
...
```

If for some reason the file can’t be opened, then the return value of `fopen` is `NULL`.

```
if (fileptr == NULL) {
    printf("ERROR: Can't open file %s to read.\n",
        filename);
    exit(program_failure_code);
} /* if (fileptr == NULL) */
```

Reading from a File

In C, we can read from a file using the function ***fscanf***, which is exactly like `scanf` except that its first argument is a file pointer, specifically the file pointer for the file to read from:

```
fscanf(fileptr, "%d", &number_of_elements);
```

Writing to a File

In C, we can write to a file using the function ***fprintf***, which is exactly like `printf` except that its first argument is a file pointer, specifically the file pointer for the file to read from:

```
fprintf(fileptr,
    "The number of elements is %d.\n",
    number_of_elements);
```

`scanf` vs `fscanf`/`printf` vs `fprintf`

What's the difference between `scanf` and `fscanf`, or between `printf` and `fprintf`?

Well, `scanf` reads from `stdin` only, whereas `fscanf` can read from any file.

Likewise, `printf` writes to `stdout` only, whereas `fprintf` can write to any file.

In fact, some implementations of C define `scanf(...)` as `fscanf(stdin, ...)`, and define `printf(...)` as `fprintf(stdout, ...)`.

fclose

The C standard library also has a function named `fclose` that takes a file pointer argument.

It closes the appropriate file and returns 0 if the file closed properly, or an error code otherwise:

```
const int file_close_success = 0;
int file_close_status;
...
file_close_status = fclose(fileptr);
if (file_close_status != file_close_success) {
    printf("ERROR: couldn't close the file %s.\n",
        filename);
    exit(program_failure_code);
} /* if (file_close_status != file_close_success) */
```

How to Use File I/O

```
...
FILE* fileptr = (FILE*)NULL;
...
fileptr = fopen(filename, "r");
if (fileptr == (FILE*)NULL) {
    printf("ERROR: Can't open file %s to read.\n", filename);
    exit(program_failure_code);
} /* if fileptr == (FILE*)NULL) */
fscanf(fileptr, "%d", &number_of_elements);
for (element = first_element;
     element < number_of_elements; element++) {
    fscanf(fileptr, "%f %f",
        &input_variable1[element],
        &input_variable2[element]);
} /* for element */
if (fclose(fileptr) != file_close_success) {
    printf("ERROR: can't close file %s after reading.\n",
        filename);
    exit(program_failure_code);
} /* if (fclose(fileptr) != file_close_success) */
...
```

Special File Pointers

- In C, there are three special file pointers that exist all the time, two of which are already old friends: `stdin`, `stdout` and a new one, `stderr`.
- We already know this:
  - `scanf(...);` means exactly the same as `scanf(stdin, ...);`
  - `printf(...);` means exactly the same as `fprintf(stdout, ...);`
- But what about `stderr`?

`stderr`

- It turns out that `stderr` is used exactly like `stdout`, except that you have to use `fprintf` to use `stderr`:
  - `fprintf(stderr, …);`
  - There's no equivalent of `printf` for `stderr`.
- Where does `stderr` go?
  To the terminal screen, just like `stdout`.
- Okay, but then why do we need `stderr` at all,
  if it behaves exactly like `stdout`,
  except less convenient to use???

Buffering I/O

- When you output to a file, there are two options:
  1. Unbuffered output: The bytes that you output go directly into the file you're outputting to, as soon as you write them.
  2. Buffered output: The bytes that you output wait in a special array in RAM until there are enough bytes to justify spinning the disk drive.

Buffer: An array where data is temporarily held, typically until a specific event occurs.

Buffering is Good! (Usually)

- When you have just a little bit of output,
  it doesn't matter whether you do buffered or unbuffered.
- When you have a lot of output,
  buffered is much faster than unbuffered,
  because you spin the disk drive less often.
  - Unbuffered: Spin the disk every time `fprintf` is called.
  - Buffered: Spin the disk only when the buffer is full.
- So, we should always buffer, right?

Buffering is Bad! (Sometimes)

- What if your program crashes while there's data in the output buffer that hasn't gotten to disk yet?
  - Lost forever and never recoverable!
  - If you don't know what your output should look like, then you might not even notice that you've lost data (which might be important data).
- So what's the most important data that you **shouldn't** buffer?

ERROR MESSAGES!

Therefore, error messages should be output unbuffered, so that they go out before the program crashes.

Why `stderr` Is Good

- `stdout`: buffered
- `stderr`: unbuffered
- All other files: buffered by default,
  unless you explicitly set them to be unbuffered.

Using stderr

```
if (number_of_elements <
    minimum_number_of_elements) {
    fprintf(stderr, "ERROR: you can't have a negative");
    fprintf(stderr, " number of elements!\n");
    exit(program_failure_code);
} /* if (number_of_elements < ...) */
```

Practical Considerations

- Opening a file or closing a file takes a long time, so don't open or close a file more often than necessary.
- But, having too many files open can crash your code, so don't keep a file open longer than necessary: don't open the file until you need it, and close it as soon as you no longer need it.
- File I/O is **VERY VERY SLOW** compared to calculations, so do as little file I/O as possible.
- **DON'T RE-READ** the same file multiple times.
- **DON'T MIX** file I/O and calculations together, because then the calculations will also be **VERY SLOW**.